LA-UR--85-1707

DE85 012733

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: DYNAMIC DATA STRUCTURES AND CONCURRENCY IN A REAL-TIME DATA ACQUISITION SYSTEM

AUTHOR(S) G. Cort, J. A. Goldstone, R. O. Nelson, R. V. Poore, L. Miller and D. M. Barrus

SUBMITTED TO Fourth Biennial Conference on Real-Time Computer Applications in Nuclear and Particle Physics
Chicago, Illinois
May 20-24, 1985

DISCLAIMER

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

DYNAMIC DATA STRUCTURES AND CONCURRENCY IN A REAL-TIME DATA ACQUISITION SYSTEM

G. Cort, J. A. Goldstone, R. O. Nelson, R. V. Poore
L. Miller and D. M. Barrus

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Abstract

We report on our efforts in developing an innovative real-time data acquisition system that makes extensive use of dynamic data structures, concurrency and state machine features. The Data Acquisition Command Language developed at the Los Alamos Weapons Neutron Research (WNR) Facility is a Pascal-based system that incorporates these features to maximize system performance, reliability and adaptability while supporting a consistent, familiar and comfortable user interface. The details and benefits of the implementation philosophy and underlying structures are discussed.

Introduction

The rapid advance of computer technology is a phenomenon that is quite familiar to any professional in the field. That these advances have been characterized by successive revolutionary improvements in the architecture, performance, functionality and availability of computer hardware is well established. The anticipation with which each new innovation is awaited, and the zeal accompanying procurement and utilization, is a testament to the professional community's dedication to improving the quality and functionality of computer systems.

Concurrent with the hardware revolution, however, software technology has made equally dramatic, if less well publicized, advances. It is well documented that achievements in the fields of language design, development strategies and methodologies, and other computer science disciplines can significantly increase the operational effectiveness of computer systems. It remains one of the great paradoxes of the technological age that although hardware advances have been accepted with great enthusiasm, incorporation of modern software technology is most often largely ignored or actively resisted.

This paper presents our experiences in developing the software for a real-time data acquisition system based upon a modern software technology. It describes our strategy for implementing a computer science approach to develop a very powerful, yet practical, software system. Benefits that are directly attributable to this approach are identified and future enhancements are suggested.

WNR/PSR Environment

In order to establish the requirements and operational constraints that led to the development of the Los Alamos Data Acquisition Command Language (DACL), a brief description of the environment in which the system must perform is appropriate. The Los Alamos Weapons Neutron Research (WNR) facility is a world class neutron scattering installation dedicated to research in physics, chemistry, materials science and biology. Operating in conjunction with the 800 MeV linear accelerator at the Los Alamos Meson Physics Facility (LAMPF), the facility supports an expanding, international user community. A major facility upgrade will result from the commissioning of the Los Alamos Proton Storage Ring (PSR) in September 1985, and will transform WNR/PSR into one of the world's premier neutron scattering centers.

This upgrade, however, will render the existing real-time data acquisition system obsolete. Its replacement, which is currently under development, will ultimately consist of a network of 8-12 computers of the VAX 11/750 class, each hosting the VMS operating system. Each computer will be dedicated to controlling the data acquisition for a single instrument or experiment. The DACL system will provide a nucleus of common data acquisition software for each computer.

In addition to supporting numerous instruments and experiments, the data acquisition environment requires that the very different needs of two classes of users (Condensed Matter Physics and Nuclear Physics) be serviced by the system. Even within a single user class, experimental configurations can vary radically. Local resources are limited to approximately four full-time programmers, so it is not feasible to build and maintain separate, customized systems or subsystems for individual instruments or experiments. The environment, therefore, encourages the development of a single generic (rather than various special purpose) data acquisition system.

Requirements and Goals

The requirements identified for the DACL system derive from the stringent constraints imposed by the system environment. These requirements are two: the system must be broadly functional to meet the needs of all users, and the system must be designed to promote efficient implementation, operation and maintenance.

The functionality and user interface requirements that were identified for the DACL system were specified using standard techniques and methodologies [1]. These requirements, and the resulting implementations, are documented elsewhere [2]. To review specific requirements or the methods employed in their determination is beyond the purview of this work.

The requirements for efficiency, stated above, derive from the broad applications to be satisfied, as well as the very limited resources that can be brought to bear on the project. Although efficiency is most generally considered within a hardware context (execution speed, resource utilization, etc.), from the perspective of the DACL project

it must encompass all project resources (especially the scarce and precious personnel resource) over the entire software life cycle. The criteria used to gauge the project efficiency include system adaptability (and maintainability), responsiveness and robustness.

An adaptable system can be made to conform to the needs of the entire range of WNR experiments and instruments without requiring intervention by system personnel. Consequently, to meet this requirement in our environment, the system must permit dynamic configuration by the user. One of the best ways to improve system maintainability is to eliminate the need for system personnel to implement frequent minor enhancements that are of no general interest. An adaptable system that is easily extended by its users promotes this goal.

In order to be responsive, the DACL system must be designed to minimize time intervals during which the user is excluded from initiating new processes from an interactive terminal. This goal is based upon the premise that the only time intervals of importance in this context are those subjectively experienced by a human operator. It should also be noted that execution speed alone is not the only criterion of importance in this regard. Indeed a system may be characterized by extremely fast execution, yet still exhibit unacceptably slow response.

For a system to be robust, it must be capable of identifying erroneous inputs or unusual conditions, recover if possible, or if recovery is impossible, degrade gracefully. Above all, a robust system protects the integrity of the experiment and associated data from an inexperienced or confused user.

Implementation Strategy

The most reliable method for incorporating the efficiency requirements described in the preceding section into a software system is to base the system design on a modern software technology. Consistent, effective tools and methodologies to support this approach are well documented [3-5]. Using these resources, the software technology can be introduced at either (or both) of two levels.

The simplest approach involves the adoption of a modern programming language. This strategy enhances programmer efficiency by providing powerful control structures, increased levels of data abstraction (through more flexible data structures), improved readability and features that simplify implementation (support for strong typing and modularity, for example). These features reduce the effort required for a programmer to implement and support an application at the code (or detailed design) level. Consequently, the most significant benefits are gained from this strategy during periods of intensive coding or detailed design activity and, to a limited extent, in performing maintenance at the code level. Certainly, such a strategy has little effect upon the specification and functional design phases, or upon the operation phase.

A more universal strategy employs computer science concepts to construct a system that is efficient across the entire life cycle. This approach introduces the modern software technology at the design level and constructs the system around it. Because the technology is integrated into the system at the top level, its scope is broadened to extend over many phases of the life cycle. The result is a system that can be efficiently designed, implemented, operated and maintained.

The DACL development project utilizes both of the above strategies to promote overall efficiency. To promote implementation phase efficiency, an extended version of the Pascal programming language is used for all coding. At the top level, the efficiency requirements described in the preceding sections are identified and are accorded the same importance during the design phase as functionality requirements. Specific computer science and information theory concepts, identified at the outset, are actively incorporated into the system design. These include the concepts of dynamic data structures, concurrent processing and state machines.

Dynamic Data Structures

To glean the greatest benefit from the introduction of dynamic data structures, these constructs should be employed at all levels of the DACL system. The choice of Pascal as the programming language makes dynamic structures immediately available at the program level. Two additional applications also present themselves, namely global control structures and data files. The DACL implementations of each of these entities is based upon dynamic data structures.

Fundamentally characteristic of any data acquisition system are its control structures. These structures are used (among other things) to define memory partitions for data histograms, to define the characteristics of hardware devices that are known to the system, to set discrimination boundaries and to store information that describes the experimental configuration or conditions. These entities traditionally reside in global, statically allocated data structures of predeclared length (usually arrays).

Within the WNR environment, such statically defined control structures present many problems. The design and implementation problems are obvious: not only must the designer foresee every control structure and substructure that is required by the system (the number of arrays), but the size (array dimensionality) of each control structure must also be adequately predicted. This usually results in overallocating each control structure to allow for "future expansion". In an environment such as ours, in which requirements vary drastically across the range of instruments and experiments supported, this policy has severe, adverse consequences. In order for all control structures to meet the needs of every experiment, each individual structure must be extensively overallocated to accomodate the demands of a worst case configuration. This results in the imposition of a large overhead (composed of the union of overheads associated with each individual configuration) upon every individual experiment.

In conventional systems, utilizing static allocation schemes, severe functionality problems also exist. Regardless of the space available in other (possibly unused) control structures, the user is strictly constrained in the number of entities of a particular class that can be defined. Extending this number for a particular control structure requires system modification and rebuild. Similar drastic action is required whenever a new control structure must be implemented. This results in a system that is functionally rigid. New applications must be postponed until required software modifications are performed, thereby stifling the spontaneity of its users. Compounding this problem, these structures are usually implemented as arrays in an obsolete language, resulting in a closed architecture that makes access by other applications difficult and prone to error.

The DACL system eliminates the above problems by implementing a locally developed facility that permits dynamic (run-time) allocation of all control structures. Owing to its functional equivalence (and analogous syntax) to Pascal dynamic variables and pointers, this facility has been dubbed the DACL heap. Indeed, from the users' perspective, the only functional difference between the DACL heap and the Pascal facility is that the DACL heap is system-global (can be accessed by any application) and nonvolatile (retains its integrity across image executions).

The fundamental component of the DACL heap is the heap record. (Here, the term record is used in the Pascal sense of a structured type that is partitioned into fields of (possibly) different types.) A heap record is defined to be 128 bytes in length, of which five bytes are required for system overhead. The DACL heap currently supports 256 differently organized heap records, of which approximately fifty are predefined by the system for the various standard control structures (CAMAC and FASTBUS hardware modules, memory control blocks, graphics structures, and status information). The remainder are available to be customized by the user community.

The DACL heap is implemented within a VMS global section that can contain 10000 heap records. Each heap record represents a single entity (an individual scaler, for example). Control structures are constructed as linked lists of similarly organized heap records within the DACL heap. Therefore, no limit exists (within the 10000 record DACL heap size) for the size of a particular control structure. Consequently, only those control structures that are required for a particular experiment are present in the DACL heap (and then only in the minimum size necessary). Control structure allocation becomes efficient and compact.

Each heap record is composed of a standard partition (the system overhead) and a control-structure-dependent partition (Fig. 1). The standard partition consists of two pointers, named NEXT and MORE, respectively, and a control structure specifier called the TAG. These three fields occupy the first five bytes of every heap record. The remaining 123 bytes are partitioned into fields according to the control structure with which the heap record is associated (as specified by the TAG value). The TAG may assume any of 256 distinct enumerated values.

The NEXT pointer is provided as a standard pointer to reference the next entity in a control structure. Although control structures may be multiply linked, they are never linked bidirectionally.

The DACL heap implementation also recognizes that the 123 bytes of the control-structure-dependent partition may not always be sufficient to completely describe an entity. In such an instance, an extension record may be defined and uniquely partitioned for the remaining information. The extension record is then referenced by the standard MORE pointer. An extension record is a distinct heap record with a standard partition (including a MORE pointer) of its own; it can also be extended. As a result, a single entity can be represented as an extensible linked list. The DACL heap also supports a dynamic string facility which allows strings of any length to be represented and eliminates the necessity of predeclaring a maximum length for any string. Figure 2 is a graphic example of a typical control structure.

The DACL heap facility provides a complete set of utility functions for the programmer/user interface. These include standard utilities to allocate and deallocate heap records as well as utilities to maintain (insert into and delete from) the control structures. Utilities for clearing the DACL heap, for listing its configuration, and for supporting the dynamic string facility are also provided. By making extensive use of recursion, these utilities can process control structures that are linked in very complex ways.

The utilities perform extensive error checking and validation to ensure that the integrity of the DACL heap is not compromised. The heap implementation is effectively hidden, and the programmer is forced to employ the standard interfaces. These interfaces protect the contents of the heap from the most error prone operations and thereby significantly reduce the possibility of data corruption due to improper access. The DACL heap is designed to provide a simple, consistent interface for non-DACL applications as well. The architecture of the DACL heap is kept deliberately open to allow access by any application. Additionally, the implementation of the heap record as a Pascal record significantly simplifies referencing information therein. Data must be referenced by record and field name, the possibility of data corruption due to incorrect offsets into a data structure is eliminated.

As has been shown, the DACL heap provides a practical alternative to traditional statically allocated control structures. The DACL heap implementation is functionally more powerful and more efficient in its use of resources than static structures. It offers the additional advantage of significantly enhancing system adaptability and robustness. By encouraging users to define and implement application-dependent control structures, it also eliminates the necessity for frequent software modifications by system personnel.

In addition to control structures, other system data structures can benefit from the introduction of dynamic characteristics. Principal among these is the structure used for offline data storage, generally a sequential file. Among the many inconveniences and disadvantages associated with a nondynamic implementation are the following.

Traditionally, the format of a sequentially organized data file is rigidly defined. In a multiple application environment, this feature has two immediate consequences. First, the format must be defined to include all possible information required by any application. This forces a file written by a particular application to contain large amounts of irrelevant information. The second consequence results from the fact that changing the file format (to accomodate a new application, for example) makes all previously written files obsolete (and unreadable). System personnel are then frequently required to modify the utilities that read and write data files, as well as to produce utilities to convert from obsolete to the (current) standard format.

The sequential organization of these files can also cause inconveniences. Access to individual entities (a single data set in a group of 100, for instance) is slow and cumbersome. Updating the file (e.g. to include processing status or previously unavailable information) is difficult.

The DACL solution abandons sequential files for the indexed sequential access mode (ISAM) organization. ISAM files permit either indexed or sequential access as required by an application. Any record in the file can be rapidly accessed using the indexed access mode. The indexed access may then be followed by a series of sequential accesses to read a large amount of information.

The DACL implementation partitions a data file into two sections: an abstract block and a data block. The abstract block contains information that describes the experimental configuration and conditions. Any information resident in the DACL heap can be transferred into the abstract block.

At run time the user can specify which heap structures are to be placed into the abstract block. This can be accomplished by accepting a default abstract block composition defined by the system or by providing a list of those control structures to be included. Standard utilities then build a unique key for each entity of each control structure specified, and write each entity to the abstract block. Keys are devised to allow the corresponding control structures to be restored from the data file to the DACL heap in the correct sequence.

The data block contains the data acquired during the run, organized by histogram name. Data histograms are decomposed into the data file in a manner similar to abstract records. Data block logical records are extremely large (32000 bytes). This enables the storage or retrieval of most histograms with a very small number of high level file accesses.

The major advantage of this approach lies in its extreme flexibility. Because the file format is dynamically defined at run time, data files contain only that information which is pertinent to the application. Data files are therefore very compact. The files are read and written without incorporating complex protocols. All files can be read and written by the same set of utilities, regardless of the information that they contain, therefore, no file ever becomes obsolete. Finally,

access to information in an ISAM file is very efficient. Any record can be located by a sequence of accesses that is comprised of a single direct (keyed) access followed by a series of sequential accesses. The necessity of reading every preceding record en route to the desired data or abstract information is eliminated. Performance is comparable to that of a sparsely populated hash table [6], but does not suffer the large resource overhead required to make hash tables efficient.

From a system perspective, the maintenance overhead associated with sequential file implementations is eliminated. The simplicity and consistency of the user interface promotes a more reliable and robust system. Overall functionality and performance are increased.

The ISAM file organization is also appropriate for implementing a catalog of data files. The DACL system maintains copies of all data files in a central archive from which users may extract desired files. By organizing each catalog entry as an ISAM record with multiple keys (file name, experimenter, date/time, instrument and title) a very powerful directory utility can be supported with minimal effort. Users can examine catalog entries for classes of runs, all runs after a specified date or by any key value. Implementing a system with equivalent functionality (but based upon sequential files) is extremely cumbersome by comparison.

Concurrency

The prime motivation for introducing concurrent processing into the DACL system is to improve the system's responsiveness. This policy is implemented in two ways: 1) by identifying classes of processes that can execute exclusively in a background mode, and 2) by identifying tasks (or components of tasks) whose completion is not required prior to the initialization of subsequent processes. The principal example of the first case is the DACL data acquisition task. The DACL data cataloging task is typical of the second.

The DACL data acquisition control task is comprised of a family of cooperating processes that execute concurrently. These include the primary data acquisition process (a detached process that executes a DCL command file which contains DCL and DACL commands) and various other processes that service it.

The DACL system is integrated into the VMS DCL command language. Consequently, data acquisition commands can be issued individually from the terminal or executed from within a DCL command file. The DACL data acquisition task provides a very powerful batch capability that permits data acquisition jobs to be executed in the background. Although this implementation is significantly more powerful than an ordinary batch facility, its purpose remains the same: to free the user's terminal for interactive processing and to allow advance scheduling of background data acquisition operations. By isolating data acquisition from inadvertent or ill-advised "live terminal" entries, overall system reliability is enhanced.

The DACL data acquisition control task supports a queue of pending runs that can be updated, modified and examined by the user. Modifications are made through a subprocess that permits the user to employ a standard text editor to perform the updates. The same subprocess also permits the user to obtain information describing the queue status, queue contents and the currently executing run. All queue operations are confined to a single VMS process (and its subprocesses). This policy allows extensive validation to be performed upon the the queue (and upon the command files referenced therein) as an integral part of the modification activity. This localization of function also simplifies the synchronization of the modification process with the process that submits runs from the queue, effectively allowing the submission process to be automatically inhibited while modifications are in progress.

The second component of the DACL data acquisition task is the run submitter process referenced above. This process is responsible for submitting the command file at the head of the pending runs queue for execution. It is present in the background (although generally in a suspended state) at all times. It is awakened whenever the conditions are right for submission of a data acquisition job, namely that no data acquisition job is currently executing and that the pending runs queue is not empty (and is not being modified). Once started, this process inhibits the modification process, submits the next command file from the queue and modifies the queue to reflect the new status. Upon exit, the modification process is enabled.

The run submitter also provides an interface between the user and the currently executing run. The user may instruct the run submitter process to kill, pause or resume the currently executing data acquisition command file.

The DACL data acquisition task incorporates a message monitor process to allow the other component processes to communicate with the terminal. This process is automatically created when the run submitter is initialized. It remains suspended until an exceptional condition occurs. The message monitor process then awakens to report the problem to the terminal. All processes thereby communicate with the terminal through a common mediator.

All three processes which comprise the DACL data acquisition task maintain status information (in the form of event flags). This information is employed by the various components for synchronization and verification. The facility is therefore implemented as a self-regulating system with high reliability.

The second major application of concurrency within the DACL system is the data file cataloging facility. This task is usually started by the data acquisition job immediately after a data file is built (although it can also be started interactively by the user). The purpose of the task is to move the data file to the archive region and to update the data file catalog.

The catalog operation is performed upon an existing file and therefore requires no information from the heap. Consequently, it is clearly

independent of other data acquisition activities and should not be permitted to delay these operations. The large amount of preprocessing performed by the cataloging task prior to transfer of the data file virtually guarantees that the delays incurred are significant, thereby encouraging implementation as a concurrently executing process.

Under this implementation the catalog facility creates a detached process to perform preprocessing, file transfer and catalog update. This allows the data acquisition job to continue while the cataloging operation is in progress. The detached process notifies the user upon completion. If an error condition occurs to prevent the catalog operation from completing successfully (e.g. no network link between the data acquisition and data archive computer systems), the user is notified and the file is queued for cataloging at the next invocation of the catalog facility. This provides sufficient protection and communication to assure reliable operation of this critical facility, and results in a significant improvement in system response.

State Machine Features

One of the major operational deficiencies of many broadly functional systems is the ability to execute syntactically valid commands in a semantically invalid sequence or context. As systems become more functional (and complex), the opportunity for committing such errors increases dramatically. The outcome can range from user inconvenience or confusion to catastrophic data loss. Because of the great expense incurred in producing the WNR neutron beam, and the high user demand for access to the facility, errors that compromise data integrity can have serious economic, political and scientific consequences.

That the various DACL features discussed in the preceding sections enhance the hardiness and reliability of particular system components is clear. However, a consistent, architectural approach is required to make the system, as a whole, more robust. The strategy adopted for DACL is to implement the entire software system as a state machine. The implementation defines four valid states for the DACL system: INITIALIZE state, RUN state, PAUSE state and HALT state. The current DACL state is maintained in a control structure in the DACL heap. At any time during execution, the DACL system is required to be in one (and only one) of these states. This approach recognizes that there are no dangerous commands, merely dangerous contexts, and that these contexts can be identified and addressed prior to command execution.

A set of valid states is defined for every DACL command. The first task of every command is to check the (current) state from which it has been invoked against its valid set. If the current state does not match one of the valid states, an error message is issued and the command is disallowed. In this manner, command execution is restricted to occur from within a sensible context, and the vast majority of semantic errors can be identified and disqualified prior to causing any damage. All command validation is done by a single utility function, allowing the facility to be simply integrated into all commands.

The possibility of inadvertently placing the system into an undesired state (as a result of a side effect of a valid command executed in an appropriate context) must also be considered. To prevent this situation from occurring, a set of state transition commands is explicitly specified. These commands represent the only means available to a user for changing the DACL state. There is one (and only one) state transition command for each valid state transition. These commands have no functionality beyond changing the DACL state. All other DACL commands are expressly forbidden to change the DACL state. The DACL state can therefore never be changed implicitly and the problem of side effects is eliminated.

Future Enhancements

Powerful as it is, the DACL implementation can also be improved. In particular, dramatic improvements in functionality and reliability could be realized simply by implementing the system in the Ada [7] programming language, an option that is within reach of many projects.

By employing Ada private types, the implementation of dynamic data structures such as the DACL heap can be securely hidden from programmers/users, thereby forcing utilization of standard interfaces for access. The architecture thereby remains open, but the system would becomes far more robust.

Generic functions (another Ada feature) can be applied to the development of many system capabilities. These constructs simplify the system by eliminating unnecessary duplication and standardizing interfaces.

The introduction of concurrency into any application is more simply accomplished if the underlying programming language supports concurrent processing. Ada incorporates concurrency as a standard language feature. The need to employ operating system features to create, control, and synchronize concurrent processes is thereby eliminated. Consequently, system implementation is simplified and portability is enhanced.

Conclusions

Although the benefits of incorporating a modern software technology into a relatively large software system (like a real-time data acquisition system) are significant, accompanying overheads also exist. Most of these overheads are associated with the manner in which such a system must be developed; they require large measures of commitment and discipline on the part of the development organization. Any project that opts to introduce a modern technology into its software must be prepared to dedicate a majority of its effort to requirements analysis and system design. In particular, the temptation to translate portions of the design to code during the early stages must be resisted. Otherwise, the technology will be incorporated in a piecemeal and haphazard fashion, the system will be fragmented and overall benefits will be minimal or nonexistent.

In the preceding sections we have described our attempts to improve the quality of our software system by exploiting the features of a modern software technology. We have described the implementation of various features including dynamic data structures, concurrency and state machine features. The advantages associated with this approach have been discussed in detail. It is our experience that this strategy has contributed significantly to the ability of a small project with limited resources to develop, implement and maintain a powerfully functional real-time data acquisition system. We contend that projects of any size can benefit from this approach.

Acknowlegements

This work was performed under the auspices of the U. S. Department of Energy.

References

[1] P. Bruce and S. M. Pederson, The Software Development Project: Planning and Management. New York:John Wiley and Sons, 1982.

[2] R. V. Poore, D. M. Barrus, G. Cort, J. A. Goldstone, L. B. Miller and R. O. Nelson, "A Data Acquisition Command Interface Using VAX/VMS DCL," presented at the Fourth Real-Time Conference on Computer Applications in Nuclear and Particle Physics, Chicago, Illinois, May 20-24, 1985.

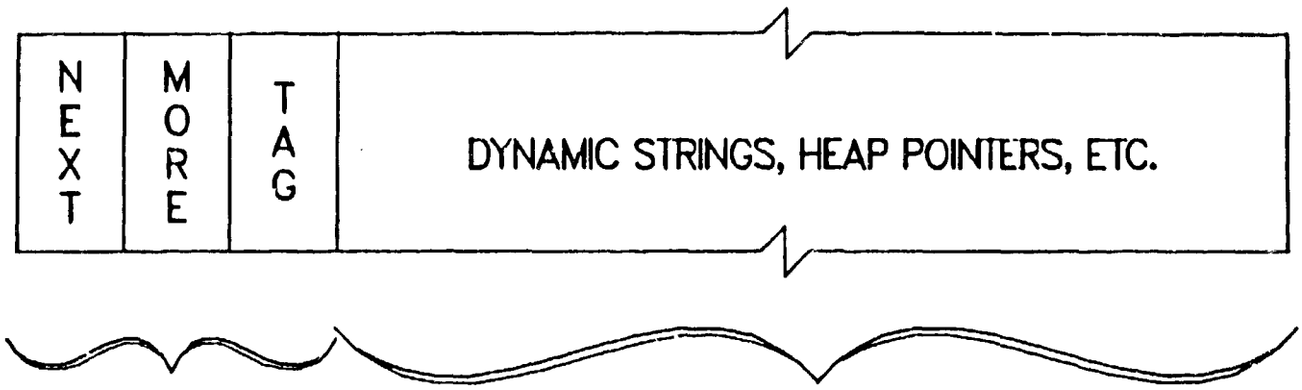
[3] R. S. Pressman, Software Engineering: A Practitioner's Approach. New York:McGraw-Hill Book Company, 1982.

[4] E. Yourdan and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1979.

[5] E. H. Bersoff, V. D. Henderson, and S. G. Siegal, Software Configuration Management: An Investment in Product Integrity. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980.

[6] N. Wirth, Algorithms + Data Structures = Programs. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1976.

[7] Ada is a registered trademark of the U.S. Department of Defense, Ada Joint Programs Office.



(a)

```
_heap_ptr = 0 .. 65535 ;

_dynamic_string_rec = RECORD
    string  : VARYING [ 16 ] OF char ;
    extend  : _heap_ptr ;
    END ;

_heap_record_tag_enum = ( . . . , _scaler, _more_scaler, . . . ) ;

_heap_record = RECORD
    more : _heap_ptr ;
    next  : _heap_ptr ;
    CASE tag : _heap_record_tag_enum OF
          .
          .
       _scaler        : ( scaler_name  : _dynamic_string_rec ;
                          scaler_crate  : integer ;
                          scaler_slot   : integer ;
                          scaler_count  : integer  ) ;

       _more_scaler : ( scaler_type  : _dynamic_string_rec ;
                        scaler_title  : _dynamic_string_rec ) ;
          .
          .
       END ;
```

(b)

Figure 1.

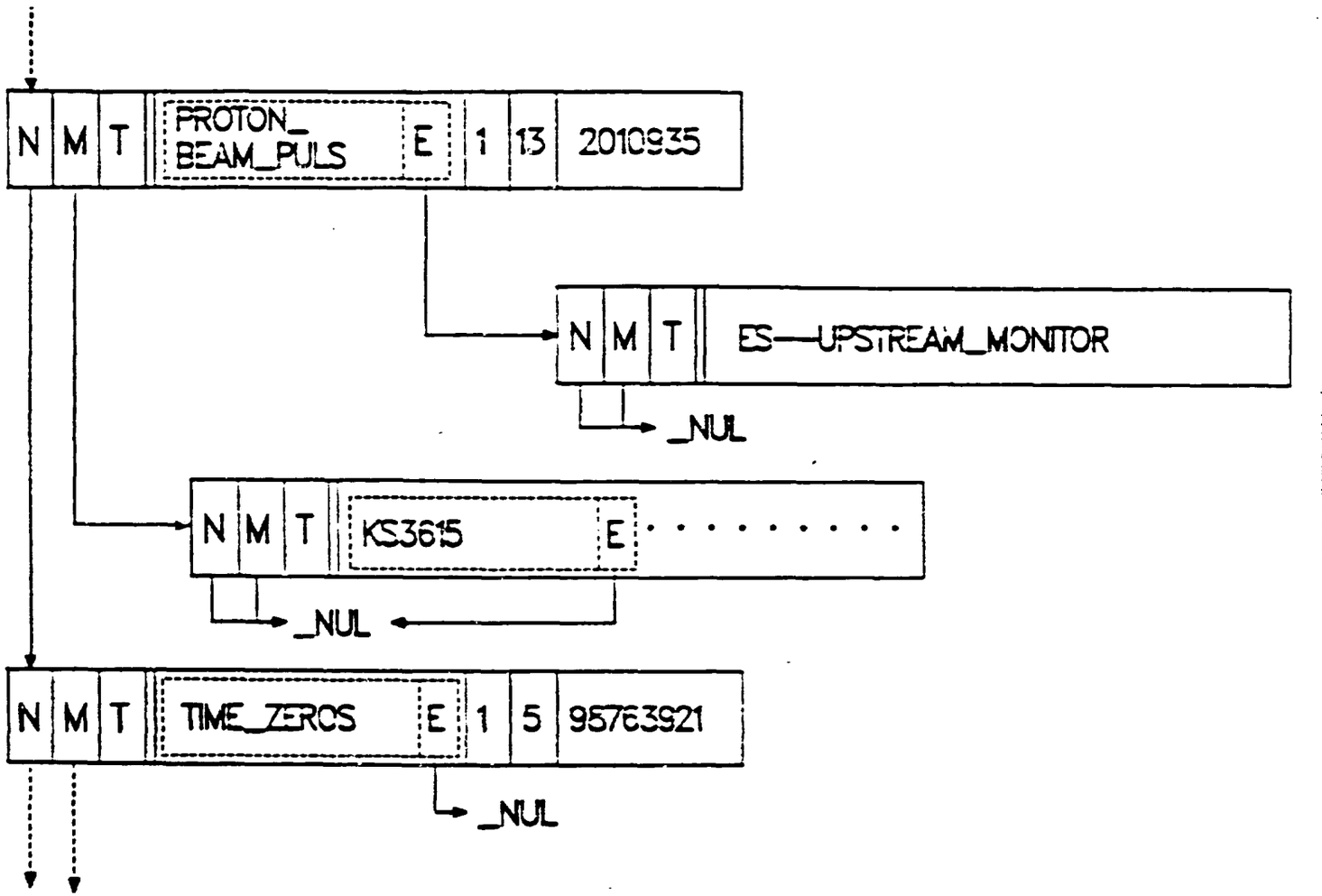

N
M
T
PROTON_
BEAM_PULS
E
1
13
2010935
N
M
T
ES—UPSTREAM_MONITOR
_NUL
N
M
T
KS3615
E
_NUL
N
M
T
TIME_ZEROS
E
1
5
95763921
_NUL

FIGURE 2.

FIGURE CAPTIONS

Figure 1. The structure of a DACL heap record. (a) Schematic representation. (b) The corresponding Pascal definition for the heap records that comprise the scaler control structure.

Figure 2. A section of a typical DACL control structure constructed from heap records. A portion of the scaler control structure (refer to Fig. 1b.) is shown schematically. Successive scalers are represented by structures linked with NEXT (N) pointers. Extension recordsare referenced through MORE (M) pointers. Dynastring extension records are referenced by STRING_EXTEND (E) pointers.